

# 爱创课堂前端培训

## JS 基础

### 第 4 天课堂笔记（本课程共 4 天）

班级：北京前端训练营 7 期

日期：2017 年 5 月 14 日

爱创课堂官网：[www.icketang.com](http://www.icketang.com)

## 目录

JS 基础.....	1
目录.....	1
复习.....	2
一、 do while 循环语句.....	3
二、 while 循环语句.....	3
三、 break 关键字.....	4
四、 continue 关键字.....	5
五、 函数.....	7
5.1 函数的声明和调用.....	7

5.2 函数的参数.....	8
5.3 return.....	9
5.4 模块化编程.....	10
5.5 函数表达式.....	12
5.6 函数的数据类型.....	12
5.7 函数声明的提升.....	13

## 复习

三元表达式：是唯一一个需要三个参数参与的表达式。

1 条件表达式 ? 值 1 : 值 2

switch 语句：特殊应用当一个值去匹配多个值时。

```
1 switch(表达式){
2     case label_1:结构体 1;
3         break;
4     case label_2:结构体 2;
5         break;          //其他的 break 必须写
6     .....
7     default:结构体 d;
```

```

8         break;           //这个 break 可有可无
9
10    }

```

for 循环语句：前测试循环语句。

```

1  for(初始化变量;最大值;步长){
2      循环语句;
3  }

```

## 一、do while 循环语句

do while 循环语句是后测试循环语句。先执行结构体，然后再判断表达式，满足条件，执行结构体。不满足条件结束循环。

语法：

```

1  do{
2      结构体;
3  }while(表达式)

```

```

1  // do while 循环语句的循环变量必须放在语句外面，不然每次都会被重置。
2  var i = 3;
3  do{
4      // do 里面书写结构体
5      i += 4;
6      console.log(i);
7  }while(i < 25);

```

do while 循环语句即使第一次不满足判断条件，也会执行。也就是说 do while 肯定会执行一次。

```

1  // do while 语句至少会执行一次
2  var i = 3;
3  do{
4      console.log(i);
5      i += 4;
6  }while(i < 3);

```

for 循环语句模拟上面这种情况非常复杂，所以当遇见结构体必须执行一次时，用 do while。

## 二、while 循环语句

while 循环语句：循环执行一段代码，直到遇见条件为假时，结束循环。

while 循环语句是前测试循环语句，和 for 非常相似。以后遇见案例可以用 while 书写，千万不要用 while 一定要用 for。

while 语句的循环变量必须写在语句外面。

```

1  while (表达式) {
2      结构体
3  }

```

```

1  var i = 3;
2  while(i < 25){
3      console.log(i);
4      i += 4;

```

```
5 }
```

```
3
7
11
15
19
23
```

```
1 // 改写为 for 循环
2 for(var i = 3 ; i < 25 ; i += 4){
3     console.log(i);
4 }
```

循环变量的自加语句，的顺序完全影响我们语句的输出。

```
1 var i = 3 ;
2 while(i < 25){
3     i += 4;
4     console.log(i);
5 }
```

```
7
11
15
19
23
27
```

while 改写为 for 循环语句非常容易。

```
1 // 改写为 for 循环
2 var i = 3;
3 for(i += 4; i < 25 ; i += 4){
4     console.log(i);
5 }
```

### 三、break 关键字

我已经找到我想到的结果，不需要再继续进行。

break 当满足遇见 break 关键字，会强制结束**当前语句**（for 循环，do while 循环， while， switch），把语句的控制权交给该循环后面的语句。

```
1 for(var i = 1 ; i < 25 ; i += 2){
2     if(i % 3 == 0){
3         console.log(i);
4         break;
5     }
6 }
```

3

在 for 循环的嵌套中, break 只能结束内层的 for 循环, 不能结束外面的 for 循环。如果想控制外面的 for 循环, 可以使用标签。给外层循环绑定一个标签, 给 break 指定标签。

```
1 for(var i = 1 ; i < 5 ; i ++){
2     for(var j = 1 ; j < 5 ; j ++){
3         if(j == 2){
4             break;
5         }
6         console.log(i , j);    //这个 break 只能管理 j,不能管理 i
7     }
8 }
```

1 1

2 1

3 1

4 1

```
1 // break 通过外层的 for 循环的标签控制
2 waiceng : for(var i = 1 ; i < 5 ; i ++){
3     for(var j = 1 ; j < 5 ; j ++){
4         if(j == 2){
5             break waiceng;
6         }
7         console.log(i , j);    //这个 break 只能管理 i,不能管理 j
8     }
9 }
```

1 1

## 四、continue 关键字

这个结果不是我想要的, 继续执行下一个。

当满足某个条件时, 有 continue 只能结束当前语句, 不会结束整个循环语句。

continue 的作用, 会终止这一次的循环, 立即进入下一次循环。

```
1 for(var i = 1 ; i < 5 ; i ++){
2     for(var j = 1 ; j < 5 ; j ++){
3         if(j == 2){
4             continue;
5         }
6         console.log(i , j);    //只能管理 j, 不能管理 i
7     }
8 }
```

```

1 1
1 3
1 4
2 1
2 3
2 4
3 1
3 3
3 4
4 1
4 3
4 4
>

```

和 break 一样可以通过绑定标签，管理指定的语句。

```

1 // 通过标签管理外层
2 waiceng : for(var i = 1 ; i < 5 ; i ++){
3     for(var j = 1 ; j < 5 ; j ++){
4         if(j == 2){
5             continue waiceng;
6         }
7         console.log(i , j); //只能管理 j，不能管理 i
8     }
9 }

```

```

1 1
2 1
3 1
4 1

```

break 和 continue 可以简化我们的代码（指的是计算机的计算过程）

break 实例：

```

1 // 用户输入
2 var n = parseInt(prompt("请输入一个正整数"));
3 // 只看这个数的开平方之前是否除了 1 还有其他的约数。
4 for(var i = 2 ; i <= Math.sqrt(n) ; i ++){
5     if(n % i == 0){
6         // 只要找到一个约数，就结束 for 循环
7         break;
8     }
9     console.log(n + "是质数");
10 }

```

continue 实例：

```

1 // 1-1000 之间所有的质数
2 // 遍历 1-1000 之间的所有数字
3 waiceng : for(var i = 2 ; i <= 1000 ; i ++){
4     // 判断 i 是不是质数
5     for(var j = 2 ; j <= Math.sqrt(i) ; j ++){
6         // 如果 j 是 i 约数，这个 i 不是我想要的。

```

```

7         if(i % j == 0){
8             continue waiceng;
9         }
10    }
11    console.log(i + "是质数");
12 }

```

## 五、函数

函数就是功能。我们可以自己封装一些语句在函数内部，函数就具有了某一种特定的功能。

function: 功能。

函数内部可以封装一段语句，这些语句是一个整体，调用的时候，这些语句要全部一起执行。

语法: function 函数名 ( ) {}

函数调用: 函数名 ( ) ;

优点 1: 帮我们简化书写，将一些重复性的语句封装起来。

```

1 // 定义函数
2 function fun(){
3     // 结构体
4     console.log(1);
5     console.log(2);
6     console.log(3);
7     console.log(4);
8     console.log(5);
9     console.log(6);
10 }
11 // 函数的调用
12 fun();

```

### 5.1 函数的声明和调用

函数的声明: 定义函数。

关键字: function。

关键字后面紧跟一个空格，在后面是我们自己定义的函数名称，紧跟着一个小括号（参数），最后是一个大括号，里面是我们封装所有语句。

语法:

```

1 function 函数名 ( ) {
2     结构体;
3 }

```

```

1 // 定义函数
2 function sum(){
3     // 结构体
4     console.log(a + b);
5 }

```

函数名的命名规则符合标识符的规则。

注意:

函数名，必须遵循标识符的定义规则。

函数只有先声明，才能够调用。

函数声明，只是告诉我们函数执行的时候有哪些语句，这里的语句并不会自己去执行。

函数要想执行，必须去调用。

调用方法：函数名();在函数名后面直接加小括号，就可以执行这个函数。

函数执行的位置，与声明的位置无关，与函数调用的位置相关。

```
1 // 定义函数
2 function fun(){
3     console.log(10);
4 }
5 // 函数的调用
6 fun();
7 console.log(1);
8 console.log(2);
9 console.log(3);
10 //函数的调用
11 fun();
12 console.log(4);
```

```
10
1
2
3
10
4
```

## 5.2 函数的参数

函数可以帮我们封装一些代码，代码可以重复调用，函数留了一个接口，就是我们的参数，可以通过参数的变化让我们的函数发生不同作用。

参数都是变量：命名规则与变量一样。

```
1 // 定义函数
2 function sum(a , b){
3     console.log(a + b);
4 }
5 // 调用函数
6 sum(1 , 2);
7 sum(1 , "2");
```

```
3
12
```

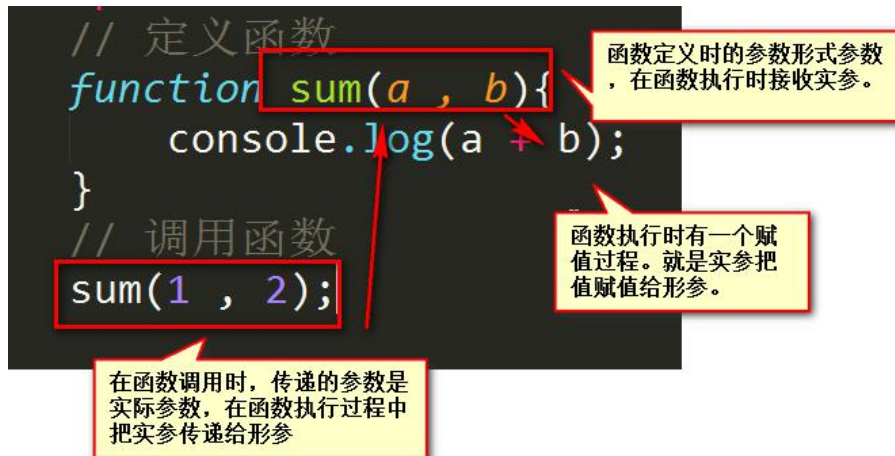
调用过程中给函数传递参数的过程就是一个给变量赋值的过程。

**形式参数**：函数定义的时候，小括号内的参数叫做形式参数（形参）。

**实际参数**：调用的时候，传递的参数叫做实际参数（实参）。

**传参**：函数执行中，有一个参数传递的过程。简称传参。





js 是一个动态类型数据语言，变量的数据类型根据里面存放的内容而变化。

实参的数据类型会影响形参的数据类型。

实参可以传任意类型的值。

```
1 function sum(a, b){
2   console.log(a + b);
3 }
4 // 调用函数
5 sum(1, 2);
6 // 实参的数据类型决定我们形参的数据类型
7 sum("12", "1");
```

```
3
121
>
```

函数的参数比较特殊，实参的个数可以不等于形参的个数。

①实参个数多于形参个数，超过形参的部分直接舍弃。

```
1 function sum(a, b){
2   console.log(a + b);
3 }
4 // 实参个数多于形参个数，多于的部分直接舍弃。
5 sum(1, 2, 3, 4);
```

②实参个数少于形参，先给前面的形参参数赋值。

```
1 function sum(a, b){
2   console.log(a + b);
3 }
4 // 实参个数小于形参个数，先给前面的形参赋值
5 sum(5);
```

```
NaN
```

因为 `a = 5`，而 `b` 没有进行赋值，`b` 会输出 `undefined`。`5 + undefined` 输出 `NaN`。

**函数的优点 2:** 函数有参数，相当于给我们提供一个 API 接口，我们可以通过接口去调用函数，执行不同的操作，后面封装函数的时候，只需要了解 API 的用途就够了，就是传参之后有什么结果，不用去了解函数里面的构造。不论是自己的函数还是用的别人封装好的函数，都只需要知道怎么用就够了。

### 5.3 return

我们函数不但可以用输出语句输出，还可以使用 `return` 语句接收我们参数。

```

1 // 定义函数
2 function sum(a,b){
3     // console.log(a + b);
4     return a + b;
5 }
6 // return 只是作为返回值，不会输出任何语句
7 // 想输出语句必须在调用时使用输出语句
8 console.log(sum(1,2));

```

```

3

```

return 关键字：在函数内部遇见 return 关键字，会立即停止 return 后面的语句，直接返回。

```

1 // 定义函数
2 function fun(){
3     console.log(1);
4     console.log(2);
5     console.log(3);
6     return;
7     console.log(4);
8     console.log(5);
9     console.log(6);
10 }
11 // 函数的调用
12 fun();

```

```

1

```

```

2

```

```

3

```

返回值将我们的函数矮化成了一个表达式。利用这个特性，我们可以将函数作为一个实际参数，传递给另外一个函数。

**函数的优点 3：**我们可以将一个函数作为返回值，传递给另外一个函数。有利于我们的模块化编程。

```

1 function sum(a,b){
2     return a + b;
3 }
4 console.log(sum(1,sum(3,4)));

```

## 5.4 模块化编程

人类从古至今，习惯将事情分工，将一些内容做成一些公共模块，模块可以重复反复使用。

模块化编程：将一些基础的公共的部分单独封装到一个函数内，可以多次被调用。

案例：输出 10000 以内的质数，模块化编程。

逆向思维的过程：输出 10000 以内的质数 → 判断是不是质数 → 找约数个数

```

1 // 书写一个函数，找任意一个数的约数个数
2 function yueshugeshu(a){
3     // 累加器
4     var sum = 0;
5     for(var i = 1 ; i <= a ; i ++){
6         if(a % i == 0){
7             sum ++;
8         }
9     }
10    // 得到约数个数
11    return sum;
12 }

```

```

13
14 //console.log(yueshugesu(8));
15
16 // 书写一个函数，判断这个数是不是质数，是质数返回 true，否则返回 false
17 function isZhishu(b){
18     if(yueshugesu(b) == 2){
19         return true;
20     }else{
21         return false;
22     }
23 }
24
25 // console.log(isZhishu(7));
26 // 输出 1-10000 之间的所有质数
27 for(var i = 2 ; i <= 10000 ; i ++){
28     // 判断 i 是不是质数，是质数就输出 i。
29     if(isZhishu(i)){
30         console.log(i);
31     }
32 }

```

案例 2:1-1000 之间的完美数，用模块化编程书写

逆向思维：1-1000 之间的完美数 → 判断一个数是完美数 → 判断一个数的约数和

```

1 // 书写一个函数，得到求他的约数和
2 function yueshuhe(a){
3     // 累加器
4     var sum = 0;
5     for(var i = 1 ; i < a ; i ++){
6         if(a % i == 0){
7             sum += i;
8         }
9     }
10    // 返回 sum
11    return sum;
12 }
13
14 // 书写一个函数，判断他是不是完美数是返回 true 不是返回 false
15 function isWanmei(a){
16     if(yueshuhe(a) == a){
17         return true;
18     }else{
19         return false;
20     }
21 }
22 // console.log(isWanmei(6));
23
24 // 书写一个函数，输出 1-任意区间的完美数
25 function shuchu(a){
26     for(var i = 1 ; i <= a ; i ++){
27         if(isWanmei(i)){
28             console.log(i);
29         }
30     }
31 }
32
33 // 调用

```

```
34 shuchu(1000);
```

**注意：**模块化编程，可以让我们的程序更加优化，各个小模块要尽量功能单一，提高重复使用率。

## 5.5 函数表达式

函数的定义可以使用关键词 `function` 来声明，还可以使用变量定义函数。是利用一个匿名函数赋值给一个变量，来定义函数。

```
1 var fun = function(){};
```

函数表达式的调用直接用函数（变量）紧跟小括号。

函数表达式的变量名，命名规则和变量一样。

函数表达式中的函数，叫**匿名函数**，也叫拉姆达函数。

```
1 function fun1(){
2     console.log(1);
3 }
4 // 函数表达式书写
5 var fun2 = function(){
6     console.log(2);
7 };
8
9 // 调用
10 fun1();
11 fun2();
```

**注意 1：**函数表达式中，函数的结尾必须写分号，而 `function` 函数不需要写。

**注意 2：**不要两种定义函数的方法杂糅

```
1 // 错误写法，不要在 function 之后在写函数名。
2 var fun3 = function fun4(){
3     console.log(3);
4 }
5 fun4();
```

```
Uncaught ReferenceError: fun4 is not defined
at 21 函数表达式.html:27
```

## 5.6 函数的数据类型

简单数据类型：number，string，boolean，null，undefined

引用类型：object, **function**，array，RegExp……

```
1 function fun1(){
2     console.log(1);
3 }
4 var fun2 = function(){
5     console.log(2);
6 };
7
8 // 数据类型
9 console.log(typeof fun1);
10 console.log(typeof fun2);
```

```
function
```

```
function
```

变量可以接受任何数据类型的赋值，函数可以给变量赋值。

简单数据类型的不同变量赋值，只是把变量的值复制一份给另一个变量。

引用类型的赋值，复制不是一个内容，而是一个指针，指针指向就是我们的函数。变量任意一个发生变化，其他的都变化。

也就是说：简单数据类型赋值的是值，引用类型传递的是地址。

```
1 // 简单数据类型只是复制的值，不互相影响
2 var a = 3 ;
3 var b = a ;
4 a = 4;
5 console.log(a);
6 console.log(b);
```

4

3

```
1 // 引用数据类型传递是地址，之间相互影响
2 var fun1 = function(){
3     console.log(1);
4 };
5 var fun2 = fun1;
6 fun2.haha = 10;
7 console.log(fun1.haha);
8 console.log(fun2.haha);
```

10

10

## 5.7 函数声明的提升

变量声明的提升：变量声明的提升，只提升定义不提升赋值。

函数声明的提升：我们可以先调用函数，后声明函数。函数的声明也是只提升函数名，不提升函数的定义。

①函数声明的提升，用 `function` 关键字定义的函数但是因为我们函数名存储的是函数的地址，所以调用函数时，函数内部的语句也会执行。

所以根据函数这个特性，我们一般是先调用函数，在最后声明函数。是结构更加清晰。

```
1 // 先调用
2 fun();
3 // 后定义
4 function fun(){
5     console.log(1);
6 }
```

1

&gt;

②函数表达式不能进行函数声明头提升。本质是将函数赋值给了变量，提升的时候，是提升的变量名。如果提前调用，值为 `undefined`，不再是函数类型，不能使用调用函数的方法。

```
1 // 函数表达式
2 // 先调用
3 fun();
```

```

4 // 后声明
5 var fun = function(){
6     console.log(2);
7 }

```



Uncaught TypeError: fun is not a function  
at 24\_函数声明头的提升.html:19

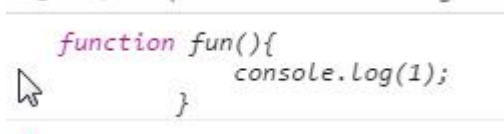
总结：防止我们引用时出现错误，要求大家在书写时，都用我们 **function** 关键字这种定义。

**函数声明提升优先于变量声明提升**。如果出现了相同的函数名、变量名，JS 预解析时，优先提升函数声明，也就是优先将这个标识符给函数使用。

```

1 console.log(fun);
2 // 定义函数
3 function fun(){
4     console.log(1);
5 }
6 var fun = 4;

```



function fun(){  
 console.log(1);  
}

```

1 // function 关键字和函数表达式
2 foo();
3 function foo(){
4     console.log(1);
5 }
6 var foo = function(){
7     console.log(2);
8 }

```



1

1

1

1

1

1

1

1

1
1
1
1
1
1